REMARKS

Applicants appreciate the thoroughness with which the Examiner has examined the above-identified application. Reconsideration is requested in view of the amendments above and the remarks below.

Applicants have amended claim 9 in response to the Examiner's rejection thereof under 35 USC § 112, ¶2.

Claims 1-5, 7-9 and 11-15 stand rejected under 35 USC § 103(a) as being obvious from Achenson et al. U.S. Patent No. 6,477,586 in view of LiVecchi U.S. Patent No. 6,427,161. Applicants respectfully traverse this rejection and request reconsideration.

Claims 1 and 11

Claims 1 and 11 recite a method of parallel processing utilizing two threads which each represent an independent flow of control managed by separate program structures. The first thread has two states: 1) processing work for the program structure, and 2) undispatched awaiting work to process. The method involves using the second thread to prepare work for the first thread to process and placing the work in a queue. If the first thread is awaiting work to process when the work is placed in the queue, the first thread is dispatched and processes the work in the queue. If the first thread is processing other work when the second thread place the work in the queue, the first thread completes processing of the other work then accesses the work and processes it from the queue.

As recited in the specification at pages 19-21, this aspect of the invention allows the first thread to be treated as a data object by the software. This first thread has a

second thread, or launcher, waiting on it. The first thread data object can be passed around and incorporated into the data structures of a program, as can any traditional data object. When desired, the software program assigns particular work to the first thread data object. The second thread places the work in the queue, and the first thread then wakes up and does. After performing work, the first thread then again waits for more work, which may be assigned from any section of the application, at any desired time. If the second thread prepares additional work while the first thread is busy, the work remains in the queue until performed by the first thread. The first thread may be reused as desired by the program structure and only destroyed after it completes a desired amount of work.

The hypothetical combinations of Achenson and LiVecchi as cited does not present a prima facie case of obviousness to one of ordinary skill in the art. Achenson discloses a multi threaded, multi processed distributed system in which different processes use a dispatcher thread for passing a remote call procedure (RCP) message to an appropriate available thread from a pool of worker threads within the process, to permit the RCP message to be processed. As the Examiner has recognized, Achenson discloses nothing about a thread having two states, and using a second thread to prepare work and place it in a queue for processing by the first thread when it is completed processing any other work.

The Examiner cites the LiVecchi patent for this disclosure. However, LiVecchi at column 3, lines 15-31 describes one implementation where a separate dispatcher thread keeps track of the status of each worker thread and, only when the worker thread is in an idle state and has no work currently assigned to it, will the dispatcher thread assign an incoming request to that worker thread. As LiVecchi recognizes, "the

dispatcher thread may become a bottleneck that prevents worker threads from being scheduled fast enough to keep all of the processors busy." LiVecchi, column 3, lines 29-31. LiVecchi solves this bottlenecking problem by the alternative system described at column 3, lines 32-50. This approach is "implemented without using a dispatcher thread." LiVecchi, column 3, lines 32-33 (emphasis added). Instead the worker threads are themselves responsible for checking a passive socket queue to determine if there are any connection requests, and if a request is waiting the thread removes the request from the queue and begins to process it. If no request is waiting, the thread then becomes an idle thread which then sleeps. Thus, in this latter system cited by the Examiner where the worker threads have two states, it is specifically disclosed that no dispatcher thread is utilized.

Accordingly, LiVecchi himself teaches away from utilizing a second thread to prepare work for a first thread that has two states and, instead, utilizes the two-state thread without a second or dispatcher thread. Thus the present invention as defined by applicants' claim 1 is not prima facie obvious since the hypothetical combination of elements, chosen as a result of the hindsight benefit of reading applicants' own specification, does not arrive at the present invention.

Claims 2-4, 7, 8 and 12-14

Dependent claims 2 and 12 recite that the second thread continues to place additional work in the queue, which is sequentially processed by the first thread as it completes processing prior work. The disclosure cited in Achenson does not make up for the teaching in LiVecchi that a two-state thread, i.e. applicant's first thread, is not used with a second or dispatcher thread. Similarly, the disclosure cited in Achenson for applicants' claims 3 and 13 (marking the work in the queue as not complete) and

claims 4 and 14 work in queue is made to wait until completed work is marked complete) does not make up for LiVecchi's teaching away from utilizing a second dispatcher thread with a first two-state thread. Claim 7 combines the subject matter of claims 1, 2 and 3, and claim 8 recites the subject matter of claim 4, so that Achenson and LiVecchi do not present a prima facie case of obviousness for these claims as well.

Claims 5, 9 and 15

Claim 5, dependent on claim 1, claim 9, dependent on claim 7, and claim 15, dependent on claim 11 add that the first thread is reused to process other work. For this, the Examiner cites from LiVecchi, "as each thread completes the work requested has been processed, it looks on the queue for its next request" (LiVecchi column 3, lines 35-37). However, this aspect of the various processes disclosed therein is from the embodiment that is "implemented without using a dispatcher thread." LiVecchi column 3, lines 32-33. Accordingly, one of ordinary skill in the art would not look to this portion of LiVecchi in connection with applicants' claimed process using a second thread to provide work in a queue to a first, two-state thread.

Claims 6, 10 and 16

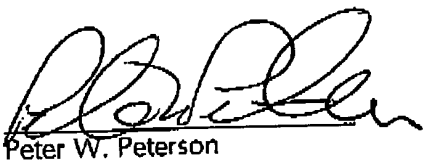
Claims 6, 10 and 16 stand rejected under 35 USC § 103(a) as being obvious from Achenson in view of LiVecchi in view of Voll et al U.S. Patent No. 6,170,018. Applicants respectfully traverse this rejection.

Applicants' dependent claims 6, 10 and 16 recite that the program structure destroys the first thread after it completes a desired amount of work. Again, applicant submits that prima facie obviousness is not established by the combination of Achenson and LiVecchi for the reasons described above, and because LiVecchi's teaching away from the method of the present invention is not rectified by any teaching

in Voll. Voll describes no two-state first thread which is fed work in a queue by a second thread. Voll is only cited for its disclosure that a thread may be destroyed when processing by it completes. This disclosure does not suggest applicants' invention wherein the first thread repeatedly cycles between the processing work state and the undispatched awaiting work state. As it receives work from a queue fed by a second thread, after which the first thread is destroyed after it completes a desired amount of work. Accordingly, applicant's invention as recited in claims 6, 10 and 16 is not obvious from any combination of Achenson, LiVecchi and Voll.

For the reasons given above, applicants submit that the claims of the instant application are in condition for allowance. Reconsideration of the rejection and allowance of the claims re respectfully requested. Any questions which may be handled by telephone should be directed to the undersigned at (203) 787-0595.

Respectfully submitted,



Peter W. Peterson
Reg. No. 31,867

DeLIO & PETERSON, LLC
121 Whitney Avenue
New Haven, CT 06510-1241
(203) 787-0595

ibmf100275000amdA

Livecchi

THREAD SCHEDULING TECHNIQUES FOR MULTITHREADED SERVERS

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to computer performance, and deals more particularly with a technique, system, and computer program for enhancing performance of a computer running a multithreaded server application. A scheduling heuristic is defined for optimizing the number of available threads. A 2-stage queue is defined for passive sockets, in order to ensure threads are not assigned to connections unless data is being sent. A new type of socket is defined, for merging input from more than one source and making that merged input available for scheduling. A function is defined for optimizing assignment of threads to incoming requests when persistent connections are used.

2. Description of the Related Art

A multithreaded application is a software program that supports concurrent execution by multiple threads—that is, a re-entrant program. A thread is a single execution path within such a program. The threads execute sequentially within one process, under control of the operating system scheduler, which allocates time slices to available threads. A process is an instance of a running program. The operating system maintains information about each concurrent thread that enables the threads to share the CPU in time slices, but still be distinguishable from each other. For example, a different current instruction pointer is maintained for each thread, as are the values of registers. By maintaining some distinct state information, each execution path through the re-entrant program can operate independently, as if separate programs were executing. Other state information such as virtual memory and file descriptors for open I/O (input/output) streams are shared by all threads within the process for execution efficiency. On SMP (Symmetric Multiprocessor) machines, several of these threads may be executing simultaneously. The re-entrant program may contain mechanisms to synchronize these shared resources across the multiple execution paths.

Multithreaded applications are becoming common on servers running in an Internet environment. The Internet is a vast collection of computing resources, interconnected as a network, from sites around the world. It is used every day by millions of people. The World Wide Web (referred to herein as the "Web") is that portion of the Internet which uses the HyperText Transfer Protocol ("HTTP") as a protocol for exchanging messages. (Alternatively, the "HTTPS" protocol can be used, where this protocol is a security-enhanced version of HTTP.)

A user of the Internet typically accesses and uses the Internet by establishing a network connection through the services of an Internet Service Provider (ISP). An ISP provides computer users the ability to dial a telephone number using their computer modem (or other connection facility, such as satellite transmission), thereby establishing a connection to a remote computer owned or managed by the ISP. This remote computer then makes services available to the user's computer. Typical services include: providing a search facility to search throughout the interconnected computers of the Internet for items of interest to the user; a browse capability, for displaying information located with the search facility; and an electronic mail facility, with which the user can send and receive mail messages from other computer users.

The user working in a Web environment will have software running on his computer to allow him to create and send requests for information, and to see the results. These functions are typically combined in what is referred to as a "Web browser", or "browser". After the user has created his request using the browser, the request message is sent out into the Internet for processing. The target of the request message is one of the interconnected computers in the Internet network. That computer will receive the message, attempt to find the data satisfying the user's request, format that data for display with the user's browser, and return the formatted response to the browser software running on the user's computer. In order to enable many clients to access the same computer, the computer that receives and/or processes the client's request typically executes a multithreaded application. The same instance of the application can then process multiple requests, where separate threads are used to isolate one client's request from the requests of other clients.

This is an example of a client-server model of computing, where the machine at which the user requests information is referred to as the client, and the computer that locates the information and returns it to the client is the server. In the Web environment, the server is referred to as a "Web server". The client-server model may be extended to what is referred to as a "three-tier architecture". This architecture places the Web server in the middle tier, where the added tier typically represents databases of information that may be accessed by the Web server as part of the task of processing the client's request. This three-tiered architecture recognizes the fact that many client requests are not simply for the location and return of static data, but require an application program to perform processing of the client's request in order to dynamically create the data to be returned. In this architecture, the Web server may equivalently be referred to as an "application server". When the server executes a multithreaded application program, the server may equivalently be referred to as a "threaded server", or "multi-threaded server".

The server is responsible for the threads. The set of threads that have been created but not destroyed will be referred to herein as a "pool" of threads. The number of threads to be created for the pool is typically specified by a user (e.g. a systems administrator), as a configuration parameter when initializing the server. Typically, this parameter is set so that the server creates a large number of threads, in order to deal with the maximum anticipated connection load (i.e. the a maximum number of incoming client requests).

The TCP/IP protocol (Transmission Control Protocol/Internet Protocol) is the de facto standard method of transmitting data over networks, and is widely used in Internet transmissions. TCP/IP uses the concept of a connection between two "sockets" for exchanging data between two computers, where a socket is comprised of an address identifying one of the computers, and a port number that identifies a particular process on that computer. The process identified by the port number is the process that will receive the incoming data for that socket. A socket is typically implemented as a queue by each of the two computers using the connection, whereby the computer sending data on the connection queues the data it creates for transmission, and the computer receiving data on the connection queues arriving data prior to processing that data.

For applications which receive requests from a number of clients, a special "passive" socket is created which represents a queue of pending client connections. Each client that needs the services of this application requests a connection to this passive socket, by using the same server port number (although communications using a secure protocol such as

Secure Sockets Layer, or "SSL", typically use a different port number than "normal" communications without security, for the same application). The server accepts a pending client connection from the special passive socket. This creates a new server socket, which is then assigned to an available thread for processing.

A number of shortcomings exist in the current approach to implementing multithreaded server applications running in this environment, which result in less than optimal performance of those applications. With the increasing popularity of applications such as those running on Web servers, which may receive thousands or even millions of "hits" (i.e. client requests for processing) per day, performance becomes a critical concern. The present invention addresses these performance concerns.

In existing server implementations, a separate "dispatcher" thread is typically responsible for monitoring the queue which receives incoming connection requests for the passive socket for a given application. To differentiate between the thread doing the dispatching, and those threads to which it dispatches work, the latter are referred to herein as "worker threads". The dispatcher thread keeps track of the status of each worker thread, and assigns each incoming request to an available thread. An "available" thread is one that is ready to run, but has no work currently assigned to it. A thread in this state may equivalently be referred to as an "idle thread". When work is assigned to an idle thread, it is no longer considered idle, and no further work will be assigned to it until it has completed its current work request. On SMP machines, the dispatcher thread may become a bottleneck that prevents worker threads from being scheduled fast enough to keep all of the processors busy.

Alternatively, a server may be implemented without using a dispatcher thread. In this approach, the threads are responsible for checking the passive socket queue to determine if there are any connection requests. As each thread completes [illegible] it first checks the queue for its next request. If a request is waiting, the thread removes the request from the queue, and begins to process it. If no request is waiting, the thread becomes an idle thread. The idle thread may then "sleep", whereby a system timer is used to cause the thread to wait for a predetermined period of time, and then "awaken" to recheck the queue to see if work has arrived. This is referred to as "polling" mode. A more common alternative to polling mode is to use event-driven interrupts. In that approach, the thread will go into the idle state and wait for a system-generated interrupt that will be invoked when work arrives, signaling the thread to become active again. Going into the idle state is also referred to as "blocking", and being awakened from the blocked state (i.e. receiving the interrupt) is referred to as "unblocking".

In current server implementations that use event-driven interrupts, as each worker thread completes its current request, it checks the passive socket queue to see if any requests are waiting. When there is no waiting request, the thread blocks. Any number of threads may be blocked at a given time. When the next incoming request arrives, an event is generated to wake up the threads. Each blocked worker thread receives this interrupt, so each unblocks and tries to take the request from the queue. Only the first worker thread will be able to take the incoming request, and the others will again find the queue empty and return to the blocked state. However, a new API (Application Programming Interface) is under development to change this approach to interrupt generation. The API is referred to herein as "accept_and_receive". According to the accept_and_receive API, when an incoming request arrives, an interrupt will be generated only to a single blocked thread.

This new interrupt approach leads to the first performance problem to be addressed by the present invention, which will be referred to herein as "over-scheduling". When the number of incoming connections is less than the number of threads in the thread pool (i.e. the connection load is less than the maximum for which the server is configured), too many threads from the pool are used to service the workload. In other words, the thread pool is being over-scheduled. This leads to inefficient use of resources.

The following scenario illustrates the over-scheduling problem. Suppose all threads are blocked, waiting for connection requests. A first request arrives. The system scheduler wakes up one of these blocked threads, and assigns the incoming request to that thread. The thread begins processing the request. Then, a second request arrives, so the scheduler wakes up a second blocked thread and assigns this new request to it. The second thread begins processing this new request. The first thread completes the request it was working on, and checks the passive socket. Finding no new connection requests there, the first thread blocks. For two requests, the scheduler has awakened two threads.

However, it may be that thread one was nearly finished with its first request at the time the second request arrived. When this is the case, it would be more efficient to wait for the first thread to finish and find the second request when it checks the passive socket, as opposed to awakening the second thread. If the scheduler awakens a new thread for each incoming request (i.e. it over-schedules the threads), a thread working on a request is guaranteed to find the incoming connection queue empty when it completes its current request and checks for another. The threads will therefore block after each completed request. The repeated blocking and unblocking operations are expensive in terms of the overall pathlength for servicing a request. When a thread blocks, the scheduler will save the context information for that thread, and the thread will move from the "ready" state to the "blocked" state. The unblocking operation requires the fairly-significant overhead associated with interrupt processing.

A further impact on the system's performance during over-scheduling is caused by the memory paging mechanism. As a thread executes, it will refer to stored information. That information must be in memory to be processed. If it is not already in memory, it will be paged in. Typically, another page must be paged out to make room for the one being paged in. Paging mechanisms use algorithms to decide which page to page out. Commonly, the least-recently-used page is selected for paging out. When over-scheduling occurs, each thread blocks after it executes, and its pages therefore become unused. The longer a thread blocks, the more likely it becomes that its pages will be paged out. Then, when the thread is awakened, its pages must be paged back in, causing another thread's pages to be paged out. The extra processing caused by these paging operations reduces the efficiency of processing the incoming request.

Additionally, the operation of checking the passive socket, only to find it empty, is a wasted operation which further reduces the efficiency of the blocking thread.

A second performance problem will be referred to herein as the "multiple input source" problem. As previously stated, a server application may receive unsecure connection requests on one passive socket, and secure connection requests on a second passive socket. This will be the case, for example, in on-line shopping applications. The client shopper may request to display available products from an on-line catalog, eventually selecting some products to be